

# **FINAL REPORT**

**Contract No. N66001-94-C-7004**  
**Contract Line Item Nos. 0001, 0002**

*Sponsored by*  
*Naval Command, Control, and Ocean Surveillance Center*  
*San Diego, California*

## **Linda® for Networks of Shared Memory Multiprocessors**

*Prepared by*  
*Scientific Computing Associates, Inc.*  
*One Century Tower*  
*265 Church Street*  
*New Haven, CT 06510-7010*

*Dr. Andrew H. Sherman, Principal Investigator*  
*Telephone: (203) 777-7442*  
*Email: sherman@sca.com*

*Report Date: January 26, 1996*

*Period Covered: December 28, 1993 – December 28, 1995*

*UNCLASSIFIED*  
*Approved for public release; distribution is unlimited.*

**20050322 138**

UNCLASSIFIED

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

This page intentionally blank.

## Table of Contents

<b>1. PROJECT SUMMARY .....</b>	<b>1</b>
<b>2. PROPOSED PHASE II RESEARCH.....</b>	<b>3</b>
2.1 RESEARCH OBJECTIVES .....	3
2.2 RESEARCH PLAN .....	3
<b>3. BACKGROUND TECHNOLOGIES.....</b>	<b>5</b>
3.1 KHOROS .....	5
3.2 VIRTUAL SHARED OBJECT MEMORIES.....	6
3.3 PARADISE .....	7
3.4 PIRANHA .....	9
<b>4. PHASE II RESEARCH RESULTS.....</b>	<b>11</b>
4.1 EVALUATION OF VISUAL PROGRAMMING ENVIRONMENTS .....	11
4.2 CONSTRUCTING THE PROTOTYPE TVPS .....	13
4.2.1 <i>The Tuple Transport</i> .....	14
4.2.2 <i>Piranha Scheduling</i> .....	14
<b>5. COMMERCIALIZATION.....</b>	<b>17</b>
<b>6. CONCLUSIONS .....</b>	<b>21</b>
<b>7. REFERENCES .....</b>	<b>23</b>
<b>APPENDIX A: PIRANHA DOCUMENTATION.....</b>	<b>25</b>

UNCLASSIFIED

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

This page intentionally blank.

## 1. Project Summary

In this project we developed a prototype visual programming system (called the Trellis Visual Programming System, or TVPS) capable of transparently supporting parallel and distributed computation on the Convex Exemplar and surrounding workstations attached on a local area network. We achieved this goal by combining Khoros<sup>®</sup> [9, 11] (a product of Khoros Research, Inc.) and Paradise<sup>™</sup> [16] (a virtual shared object memory product of Scientific Computing Associates, Inc.<sup>1</sup>). The TVPS has the following major features:

- It provides a two dimensional visual programming language interface (i.e., the Cantata portion of Khoros [18]) in which boxes (called "glyphs") are used as graphical representations for entire programs, and output-to-input data flows are depicted using lines between glyphs.
- It automatically and dynamically assigns computations corresponding to Khoros glyphs to available processors at run time in order to exploit available computing resources and thereby minimize the time to completion. Heterogeneous collections of processors are fully supported.
- It allows the use of parallel implementations for individual glyph computations, and it provides underlying infrastructure for interprocess communication within such parallel glyphs. It does not, however, provide for transparent transfer between corresponding processes of successive parallel glyphs.
- It allows the pool of available processors for glyph execution to expand and contract dynamically at run time, reflecting changes in processor availability and usage, as well as relative job priorities.
- It facilitates the creation of Khoros programs capable of continuing to run even in the presence of processor or network failures, requiring only that the computational modules handle the impact of side effects that are not visible at the level of a Cantata visual program.

The prototype TVPS was developed for Hewlett Packard's PA-RISC architecture and has been run on HP 9000 workstations and a Convex Exemplar at NRad in San Diego using version 2.0.1 of Khoros. It has also been run on Sun workstations.

In the main body of this report, we provide a detailed discussion of the research and development activities leading to the prototype TVPS. In addition, we describe the steps already taken towards identification and exploitation of commercial opportunities based on the TVPS.

---

<sup>1</sup> Paradise and Piranha are trademarks, and Linda is a registered trademark, of Scientific Computing Associates, Inc. Other trademarks used herein are the properties of their respective owners.

UNCLASSIFIED

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

This page intentionally blank.

## **2. Proposed Phase II Research**

### **2.1 Research Objectives**

As stated in Deliverable A001 to this contract, the principal technical objective for the Phase II project was to develop a prototype Trellis Visual Programming System based on one of the commonly used scientific workbench systems (Khoros<sup>®</sup>, AVS<sup>®</sup>, Data Explorer<sup>®</sup>, and Iris Explorer<sup>®</sup>). The prototype TVPS was to have the following features:

1. It would have a two dimensional visual programming language interface.
2. It would automatically assign computational nodes to available processors in order to minimize the time to completion.
3. It would allow parallel computational tasks as individual nodes and integrate their parallel execution into the total computation in a seamless way.
4. It would allow the pool of available processors for parallel computation to expand and contract as warranted so as not to compromise the interactive users of the processors.

As will be clear from further discussion, we have met these goals in the prototype TVPS.

### **2.2 Research Plan**

As stated in Deliverable A001 under this contract, our plan was to begin development of the Trellis Visual Programming System by first identifying which of the conventional workbench environments to extend. The leading candidates were Khoros, AVS, Data Explorer, and Iris Explorer. Our strategy was then to integrate the workbench code(s) with SCIENTIFIC's existing technology to produce a prototype TVPS. We then planned to extend and thoroughly test this prototype TVPS.

The main tasks for the project were:

1. to extend the visual programming language, most likely from one of the four standard workbench systems mentioned above, to include a structure for generalized data flow;
2. to develop a scheduler to dynamically map TVPS computational nodes onto available processors for parallel execution;
3. to develop appropriate interfaces to permit the use of parallel C-Linda programs as TVPS computational nodes; and
4. to investigate the use of Piranha-like ideas [6, 10] for the TVPS to allow the collection of processors allocated to a TVPS session to dynamically expand and contract as availability warrants.

UNCLASSIFIED

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

This page intentionally blank.

### 3. Background Technologies

This project builds on two major component technologies: a visual program development and execution system (Khoros), and a virtual shared object memory system (Paradise) that includes an adaptive, dynamic scheduler for parallel and distributed applications (Piranha). This section provides some basic background on these technologies.

#### 3.1 Khoros

Khoros is a software integration and development environment that includes a number of components:

1. a visual programming language (Cantata [18]),
2. a suite of software development tools that extend Cantata, making it easier to create new applications, including an interactive user interface editor,
3. a runtime execution system, and
4. packages for interactive image display, 2D/3D plotting, and a large variety of image processing, data manipulation, scientific visualization, geometry and matrix operations.

While Khoros's programming services and software development tools were intended originally to support development of engineering and scientific applications, it is just as natural to apply them to nontechnical commercial applications as well. Applications written in Khoros can take advantage of the same capabilities exploited by Khoros's own data processing and visualization routines, including the ability to transparently access large data sets distributed across a network, operate on a variety of data and file formats without conversion, and maintain a consistent presentation with a standardized user interface. The software development environment provides developers with a direct manipulation graphical user interface design tool, automatic code generation, standardized user interface and documentation, and interactive configuration management. The Khoros software development system can also be used for software integration, where existing programs can be brought together into a consistent, standardized, and cohesive environment.

All information processing and visualization programs in Khoros are available via the visual programming language, Cantata [18]. Cantata is a graphically expressed, data flow visual language which provides a visual programming environment within the Khoros system. In Cantata, a visual program is described as a directed graph, where each node (called a "glyph") represents an operator or function and each directed arc represents a path over which data flows. This flow-chart-like approach is a natural environment in which to describe applications, particularly those that are based on coarse grain distributed processing. Khoros's visual hierarchy, iteration, flow control, and expression-based parameters make it a powerful simulation and prototyping system for complex applications. It is widely used for just this purpose by the Department of Defense and defense contractors.

In this project, most of our attention was focused on two particular aspects of Khoros: its scheduling service and its data transport service. The standard implementation of Khoros includes a scheduling daemon (*phantomd*) that determines when glyphs are ready to run and then schedules the glyph on the local machine. In principle, *phantomd* supports general distributed or parallel execution, but only a very limited capability is available in the released version of Khoros (as of version 2.0.1). One goal of this project was to provide a simple, yet very general and flexible means of executing Khoros programs in heterogeneous parallel and

distributed computing environments using a dynamic scheduler based on existing COTS products from SCIENTIFIC.

The standard Khoros inter-glyph data transport mechanism uses files. One glyph writes its output to a file, and the next glyph reads its input from the same file. This can be used to support distributed computing, though care is required in heterogeneous environments, and it may require special file system configuration to make sure that files are accessible where they are needed. Another goal of this project involved replacement of this file-based data transport mechanism with one that would be easier to configure and use.

### 3.2 Virtual Shared Object Memories

The software tools for parallel and distributed computing used in this project are based on a memory model for multiprocess computing that generalizes the model made popular by Linda<sup>®</sup> [1, 4, 15]. The model is based on the use of virtual, associative, logically-shared object memories (VSOMs) that contain collections of logically-ordered sets of typed data, known as "tuples." Computational processes accomplish work by generating, using, and consuming tuples.

The sorts of VSOMs supported by Linda and Paradise are tailored specifically to the needs of parallel and distributed software ensembles by accommodating both data sharing and inter-process coordination. These memories store not bytes, but complete multidimensional data objects. Three basic access operations (*out*, *in*, and *rd*) are provided instead of the two (*read* and *write*) that are provided by conventional address spaces, and these operations have built-in synchronization. Data transfer between machines is implicit and demand-driven, based on actual usage.

A key feature of SCIENTIFIC's VSOMs is that they are associative memories. Tuples have no addresses; they are selected for retrieval on the basis of combinations of their field types and values. Thus the five-element tuple (*A*, *B*, *C*, *D*, *E*) may be referenced as "the five element tuple whose first element is *A*," or as "the five-element tuple whose second element is *B* and fifth is *E*" or by any other combination of element values. (Matching is sensitive to the element types, as well as their values.) Formal parameters (or "wild cards") are used in selection templates to designate "output fields" that enable values to be transferred from VSOMs into local variables.

Apart from their basic virtues, VSOMs provide a superior environment for program construction and execution. Message passing libraries such as MPI [7, 13] and PVM [5, 17], and generic shared address spaces with locks, are really sets of low-level operations; they are not programming methods. A VSOM, on the other hand, provides operations and a programming method as well — in brief, algorithms plus distributed data structures making distributed or parallel programs.

Programming models based on VSOMs have another, less obvious advantage as well: language independence. In general, the VSOM and its associated operations are orthogonal to the base programming language — essentially the same syntactic forms work with FORTRAN, C or any other similar language. As a result, parallel programming techniques also become language independent, and programmers can quickly and easily transfer their parallel programming skills to new situations without having to learn entirely new parallel languages.

Many computing paradigms are well supported by VSOMs; here are just two important examples(cf., [2, 3]):

1. *Task bags and adaptive parallelism.* "Task bags" are a representation of sets of independent tasks that can be executed concurrently, in any order. With a VSOM, one would implement a loop in which task descriptors were dumped into the VSOM, there to be claimed by worker processes on an as-available basis. The extent of code modification from a corresponding original serial program is quite modest.

A desirable feature of a VSOM-based task-bag paradigm (and one exploited by the TVPS) is that it can be executed adaptively: the number of participating machines may change during execution without affecting correctness. A runtime system such as SCIENTIFIC's Piranha system can transparently, dynamically and adaptively schedule and deschedule machines based on any desired set of criteria (such as load, capabilities, physical presence, job priorities, etc.). When tasks are descheduled, the VSOM can be used to store state information (in essence redepositing a partially-computed task result); in addition, programmers can specify "retreat" procedures that are executed when descheduling occurs. These features make the VSOM-based approach a natural fit to the sort of computational environments targeted in this work.

2. *Ensemble Parallel-Distributed Applications.* The VSOM model is especially well suited to client-server systems, the most common form of distributed application. For example, a parallel application on a dedicated multiprocessor might want to display data on a separate graphics workstation. A VSOM provides a perfect medium for handling the sort of asynchronous, often unpredictable data transfer required in this case. Using SCIENTIFIC's Linda/Paradise environment, even the operations that allow the processes of the parallel application to share data are exactly the same ones that allow the processes on the multiprocessor to share data with the graphics workstation.

A great strength of the VSOM model is its explicit support for distributed data structures, i.e., data structures that are uniformly and directly accessible to many processes simultaneously. Any tuple sitting in a VSOM meets this criterion: it is directly accessible to any process using that VSOM. Thus, a single tuple constitutes a simple distributed data structure, but it is easy and often useful to build more complicated multi-tuple structures (arrays, queues, or tables, for example) as well. The TVPS system built in this project uses just this sort of generality to replace the Khoros file transport system by building an ordered inter-glyph data stream.

Another feature of the VSOM model is its intentionally loose coupling among processes. Processes interact only with the intermediation of data stored in a VSOM. As a result, it becomes very simple for processes to interact without knowing with whom they interact and without any presumption of simultaneous execution. In the TVPS system, this translates into the fact that the inter-glyph data can outlive the producing (upstream) glyph process to be used as input by a consuming (downstream) glyph process that may not yet have been created and whose physical location need not even be determined until it is created.

### **3.3 Paradise**

Paradise is SCIENTIFIC's most flexible and powerful parallel and distributed computing environment. It is based upon the VSOM model, and it is capable of providing one or more persistent VSOMs to be shared among multiple applications. The programs sharing these common VSOMs remain completely independent, potentially running at different times or on vastly different kinds of computers.

Paradise provides a great number of benefits for parallel and distributed computing, many of which are particularly useful for the TVPS:

- It provides for multiple VSOMs that may be entirely independent of the applications that create and use them. In fact, Paradise VSOMs may be *persistent*, so that they can continue to exist even after their creating programs terminate. The TVPS uses a separate VSOM for each output line in a Cantata visual program.
- It can connect existing programs quickly and easily. Completely independent programs can share data with only minimal modifications. An important consequence of this fact is that programs with different functions can be developed independently. When one program changes, it need have no effect on any others with which it happens to share data. Paradise brings modularity at the application level to ensemble computing, an important feature for the TVPS, which needs to honor fully the Khoros commitment to standardized inter-glyph interfaces.
- It supports dynamic attachments to shared data spaces. Programs sharing data need not run concurrently, and applications can detach from and reattach to VSOMs as desired. This means it is easy for the TVPS to schedule processes where and when it wishes, without having to worry about getting data connections set up properly.
- It allows for transparent heterogeneity. Programs running on distinct, or even incompatible, computer architectures can access a common VSOM without any special handling. Paradise takes care of all necessary data conversion automatically using XDR. In the context of the TVPS, this means it is straightforward for different glyphs to be executed on different machine types.
- It is based on the same underlying technology as SCIENTIFIC's Linda, so it can be combined with Linda to provide a single approach for both parallel and distributed computing. Of course, any other approach for building parallel glyph modules (such as message passing libraries like MPI or PVM, vendor-specific tools, or raw shared memory) could be used as well, but the combination of Linda and Paradise in the context of the TVPS is particularly natural and consistent architecturally.
- It offers a secure distributed computing environment. Paradise includes facilities for controlling access to sharable VSOMs above and beyond those provided by the host computer systems. This means that multiple TVPS applications can run concurrently without interference.
- It can bring indirect parallelism to sequential applications. Paradise can be used to create so-called "live libraries" — servers which perform various standard types of computations for any client which requests them. Unlike standard subroutine libraries which have been parallelized for a single type of computer, such servers can run in parallel on any available computer system, allowing, for example, a sequential application on a mainframe to take advantage of a specialized multiprocessor or the idle CPU capacity in a local area network to perform computationally-intensive operations. In this way, available computing resources can be dynamically deployed on an enterprise-wide basis. This sort of capability could be used to support a parallel Khoros toolbox (though that has not been to date in this project).
- It provides for fault-resilient execution. Paradise includes a sophisticated fault resilience facility which allows programs to guarantee data integrity despite hardware failures, abnormal process termination, lost connections between computers, and other such failures. Fault resilience is structured around a *begin-commit* transaction strategy similar to that used by state-of-the-art database systems. Changes — atomic transactions in database parlance — to a VSOM are not made permanent until they are committed by the

application performing them, and uncommitted transactions are canceled in the event of failures. In addition, Paradise provides an automatic file backing mechanism. The combination of these fault resilience features make it possible to build reliable Khoros applications using the TVPS.

### 3.4 Piranha

In our discussion of the task-bag paradigm above, we referred briefly to the concept of adaptive execution of parallel programs. For the Linda and Paradise environments, this is accomplished through the use of Piranha [6, 10]. Although the (theoretical) Piranha model is completely general and makes no assumptions about how programs are organized, the SCIENTIFIC implementation used in the TVPS exploits many of the features of Paradise, including the VSOM model and the operations used to access the VSOMs. Appendix A to this report contains a variety of documentation about Piranha, including two UNIX *man* pages and a section from the Paradise User's Guide [16].

The central design goal of Piranha is the harnessing of all the CPU cycles available on multiprocessors or on workstations within a local-area (or even wide-area) network. It is intended to be highly flexible, allowing the set of processors participating in the program execution — the *piranhas* — to expand and contract during the course of a run, in response to the changing usage levels on the individual processors, to the relative importance of jobs competing for execution, and to the availability of specific processors, among other things. For example, if a processor becomes free after a Piranha program has started, it can still join in the execution. On the other hand, if the load on a participating processor should increase during a run, execution of the Piranha program on that processor can terminate without disturbing the other processors or affecting the correctness of the final program results.

Piranha programs are structured around three key routines:

- *feeder*, which runs on a designated home processor and oversees the entire computation;
- *piranha*, which causes the real work to be done in the parallel computation; and
- *retreat*, which is called whenever a Piranha process must terminate in mid-execution because the processor on which it is running needs to be reassigned.

In the context of the TVPS, the *feeder* controls the overall execution of a Khoros program, creating tasks that correspond to glyph computations that are "ready to run." The *piranha* is simply a UNIX routine that causes the proper glyph executable program (typically a sequential program) to be executed (via *fork/exec*) on whatever processor is running that copy of the *piranha*. *Retreat* simply resets the state of the input and output tuple transports as if the corresponding glyph had never started execution.

UNCLASSIFIED

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

This page intentionally blank.

## 4. Phase II Research Results

The research in this project was divided into two major parts:

1. Evaluation of the various visual programming environments, eventually leading to the selection of Khoros; and
2. Construction of the TVPS, including basic research on Paradise and Piranha and integration of those technologies with Khoros.

### 4.1 Evaluation of Visual Programming Environments

In the first stage of the Phase II project, we investigated a number of visual programming environments for use in the TVPS. The four major candidates were:

1. Khoros, a product of Khoros Research Inc.;
2. AVS, a product of AVS, Inc.;
3. Data Explorer, a product of IBM; and
4. Iris Explorer, a product of Numerical Algorithms Group, Ltd., based on a product of the same name of Silicon Graphics, Inc.

All of these products were originally designed for purposes other than serving as general visual programming environments, although all have many of the features desirable for such a use. Khoros was originally intended for specifying data flow programs for image analysis and signal processing applications, though it included a visual programming front end (Cantata) designed to support the construction of data flow applications. AVS, Data Explorer and Iris Explorer were all developed as high-end visualization tools: they support a data flow paradigm similar to Khoros's, but their basic assumption is that the computational modules (called "glyphs") perform visualization-related operations rather than general purpose computation.

In Figure 1 we compare the most important features of the four products. For each feature, we

Feature	Khoros	AVS	DX	IX
Generality of control flow (Front End)	++	-	--	+
Generality of data access (Back End)	0	-	--	-
Support for distributed execution	+	+	+	+
Extensibility	++	+	0	+
Availability & stability of source code	+	-	-	--
Availability of "developer support"	--	-	0	-
Portability/Platform coverage	++	++	0	--
Product Support	--	+	0	0
Value to NRaD	++	?	?	?-
Commercial potential for SCIENTIFIC	--	++	0	-
Priority for Project	++	++	0	-

Figure 1: Comparison of Visual Programming Environments

have assigned a rating between "--" (very poor fit or unsuitable for this project) and "++" (extremely good fit for this project), based on our perception of the product's suitability for this project.<sup>2</sup> The features considered were:

- **Generality of Control Flow:** Typical data flow programs do not allow for loops or conditionals. The rating for each of the products reflects their explicit support for such program constructs, which are important for development of general parallel and distributed programs. While Khoros was clearly the best at the time we evaluated the products, Iris Explorer was undergoing revision intended to enhance its control flow capabilities.
- **Generality of Data Access:** None of the products provides for particularly general and flexible access to data passed between glyphs. However, the three visualization-based products (and especially Data Explorer) make the presumption that all glyph computational modules fit into very rigid interfaces suitable for visualization purposes, and they make it quite difficult to adapt to non-visualization environments. Khoros does not impose such severe restrictions on data access, but it is clearly designed with the intent that data is to be managed by low-level internal routines and is, in general, not to be accessed by non-KRI routines except through well-specified interfaces.
- **Support for Distributed Execution:** All four products make it possible, though not easy, to run some glyph computations on one machine and some on another. The TVPS makes such distributed execution entirely transparent, which is a significant enhancement over any of the four products.
- **Extensibility:** All of the products make it "possible" to add new glyph modules, though the ease of doing so varies depending on the rigidity of the module interfaces. Khoros is clearly the most flexible, while Data Explorer requires modules to fit into a small number of different classes, each with a very rigid, visualization-oriented interface.
- **Source Code Availability:** Khoros source code is publicly available (though it is not in the public domain). All the others are commercial products, and the vendors view the source code as highly proprietary. We were unsuccessful in negotiating arrangements for source code access with any of those vendors.
- **Developer Support:** This rating reflects our assessment of the ease with which we would be able to obtain support for or assistance in possible changes required in the base product to support the extensions required for the TVPS. Basically, none of the three commercial developers expressed interest in providing much help, and the Khoros product was explicitly supported only as a research product (though that may have changed since we made this assessment).
- **Portability/Platform Coverage:** Khoros and AVS have been used on a wide variety of UNIX platforms, so, in principle, at least, they would be suitable for a heterogeneous distributed environment. Data Explorer, while originally oriented entirely to IBM's own machines, has been extended to a few workstations as well. Iris Explorer, at the time of the assessment, was to be available only on SGI machines (from SGI itself) and Sun workstations (from NAG).
- **Product Support:** No support was available for Khoros (though there was and is an active on-line news group in which users provide assistance to one another, with occasional

---

<sup>2</sup> We would remark that suitability or unsuitability for use in this project is unrelated to the quality of the product for its intended purpose. We made no effort to evaluate the products for their intended purposes.

input from Khoral Research). The other products are commercial offerings, and the vendors offer varying degrees of customer support.

- **Value to NRaD:** This represents our assessment of the fit between each product and the NRaD efforts. To our knowledge, only Khoros has a significant user base at NRaD, while the other products are either unused or used by a very small number of users.
- **Commercial Potential for SCIENTIFIC:** It appeared to us that AVS was the most suitable product to serve as the basis for a commercial version of the TVPS. This was based mainly on the fact that it has a widespread and committed user community, as well as on its portability. Iris Explorer has historically been restricted to SGI platforms, and it is unclear whether it will make a successful transition to a broader machine base. Similar comments apply to Data Explorer, though IBM had already released it for a number of non-IBM workstations at the time of our assessment. Finally, Khoros was purely a research and custom application environment at the time of our assessment, though that has changed recently as Khoral Research has begun to pursue commercial opportunities.

The final line of the table in Figure 1 is an assessment of the priority of each product for this project. The priority of Khoros was high due to its importance to NRaD and the Department of Defense more generally. The priority of AVS was high due to its commercial potential; subsequently, however, we were unable to reach a suitable development agreement with AVS, Inc., so we based the TVPS entirely on Khoros.

#### 4.2 Constructing the Prototype TVPS

The prototype TVPS is a version of Khoros to which two major, independent new features have been added as options. At run time, UNIX environment variables can be set to control which, if either, of the new features is used.

- A new inter-glyph data transport system (called the *tuple transport*) has been added as an optional replacement for the standard file transport system; and
- A dynamic glyph scheduler based on Paradise Piranha™ technology has been added as an optional replacement for the Khoros *phantomd* scheduling daemons.

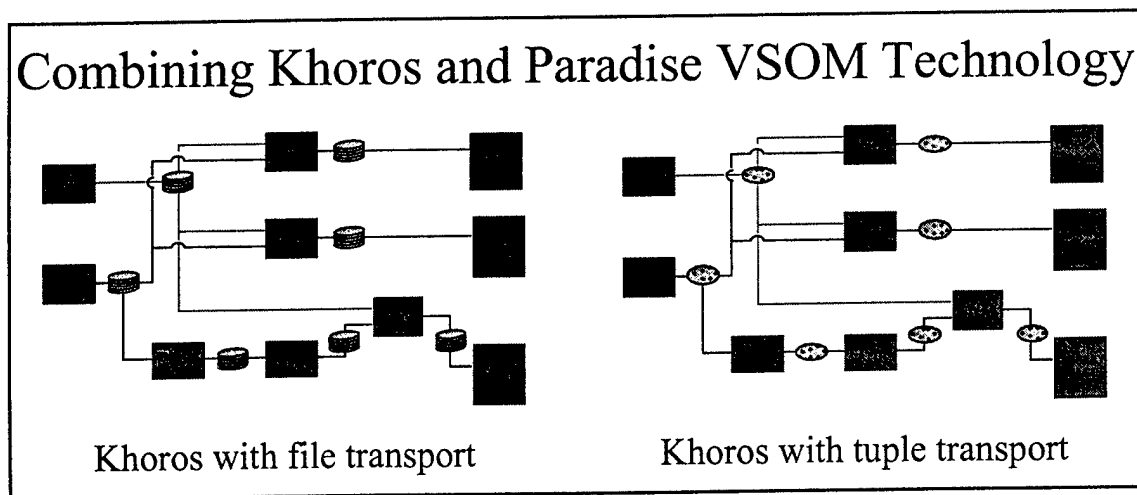


Figure 2: Data Transport Comparison

Both changes exploit virtual shared object memory (VSOM) technology and require the presence of a Paradise system to provide run-time support for the VSOMs. Figure 2 compares the original Khoros design with that of the TVPS with respect to the way that data moves through the application. It should be emphasized that the changes are completely independent of one another: the parallel scheduler and the tuple transport can be used together or separately, depending on the particular computing environment.

The TVPS relies on Khoros and its visual programming interface Cantata to provide all of the interface visible to the user. Under the covers, the TVPS uses Paradise VSOMs and the Paradise/Piranha run-time system to provide the parallel functionality. To most Khoros users, the changes should be transparent; existing Khoros applications should run normally, except that they will now run in parallel.

#### 4.2.1 The Tuple Transport

In the prototype TVPS, a separate virtual shared memory is used for each output line in the Khoros visual program. Within each such VSOM, the data are organized as a linked list of tuples containing a stream of data in what amounts to a virtual file. Each VSOM is actually stored in the virtual memory of a Paradise server,<sup>3</sup> so the data can be accessed using standard Paradise operations. The data transport layer of Khoros was modified to perform all the usual file operations (e.g., seek, rewind, read, write, etc.) by manipulating the tuples in the VSOMs.

In theory, the tuple transport has a number of advantages:

1. The VSOMs are accessible from any machine that can "see" the Paradise server (e.g., any workstation on the same LAN, or any processor on the Exemplar, in the NRaD case). This is completely independent of the configuration of NFS or any other remote file access system. In addition, it means that machines with incompatible file systems (such as PCs running Windows NT and UNIX workstations) can share data.
2. Since each VSOM is kept in virtual memory by a Paradise server, data access should be somewhat faster than with true disk I/O (though this may depend on the characteristics of whatever remote file system might be in use for the disk I/O).
3. The data in the VSOMs is accessible by any Paradise program, so it is straightforward to share the data among several independent programs concurrently. This makes it especially easy for several downstream glyphs to share the same upstream output data. In addition, this observation suggests a simple strategy for supporting parallel glyph programs by having them share information through Paradise VSOMs (though we did not attempt this). It also offers the possibility of creating system monitors that keep track of the state of the overall Khoros application by observing the contents of the interglyph VSOMs.

In the course of this project, we did not have time to run significant benchmarks or tests to verify these advantages in practice. However, significant experience with Paradise in other settings leads us to believe that all actually do hold.

#### 4.2.2 Piranha Scheduling

In order to support parallel executions of Khoros visual programs, we developed a modified version of the standard Piranha system that is included in SCIENTIFIC's Paradise system.

---

<sup>3</sup> Each VSOM is stored and managed by exactly one Paradise server; but there may be several independent Paradise servers, each of which is responsible for a subset of the total collection of VSOMs.

The modifications were designed to cater to several significant differences between Khoros applications and traditional Piranha programs:

1. Each glyph module, which becomes a Piranha task in the parallel Khoros, is a complete program. In other uses of Piranha, a Piranha task is merely one function in a larger Paradise program.
2. The glyph modules are created without any knowledge of or support for Paradise. In ordinary Piranha programs, the Paradise compilation system has been used, so it is possible to insure that various support routines for Paradise and Piranha are present.
3. Only one copy of each glyph module is permitted to execute at any given time (a Khoros restriction that might be relaxed, as we discuss later). In Piranha, several copies of the task function can execute concurrently.
4. Except in the case of failures, Khoros glyph modules are always permitted to run to completion without retreating, whereas Piranha tasks may retreat at any time without warning. The reason for this restriction on glyph executions is that glyph modules are built as complete programs, without any requirements that they be restartable. Retreating such a program might cause side effects (e.g., partially completed I/O) that could not be handled properly. When failures occur, however, the modified system does restart the glyph module in an effort to move the entire application towards successful completion.

Incorporating the modified Piranha system into the TVPS was straightforward. Khoros scheduling is data driven. Any glyph module is considered "ready to run" whenever it satisfies two criteria:

1. Its previous execution (if any) is complete.
2. Complete, new inputs are available for each of its incoming data lines.

In the TVPS, each time Khoros declares a given glyph as "ready to run," a Piranha task is generated that will execute the corresponding glyph program. The Piranha run-time system (a loose-knit collection of cooperating user-level daemons running on available processors) then schedules the task on the first available suitable processor. The glyph executable itself is unchanged; only the Khoros interface routines have been modified to go to the appropriate VSOMs to retrieve input or create output. Paradise transactions are used automatically by the Piranha system to make certain that failed glyph executions (due, for example, to processor failures) are restarted with proper input. (However, side effects arising from partial execution are not handled in the prototype system.)

Apart from the obvious advantages of parallel/distributed execution, dynamic scheduling, and some degree of fault resilience, we believe that the use of Piranha scheduling in the TVPS can offer a significant portability advantage. Specifically, one of the most frequent difficulties with Khoros is porting it to new machines and/or operating systems. Much of the difficulty lies in the graphical interfaces, file system issues, and the scheduling daemons. With the TVPS, however, Khoros programs can exploit a wide range of machines, provided only that Paradise, Piranha and the glyph programs (as opposed to Cantata and the Khoros system itself) run on the machines. Paradise and Piranha already run on a large number of UNIX machines and on personal computers running the Windows NT operating system, and we expect that porting glyph programs should be far easier than porting the entire Khoros system itself (which contains many thousands of lines of code).

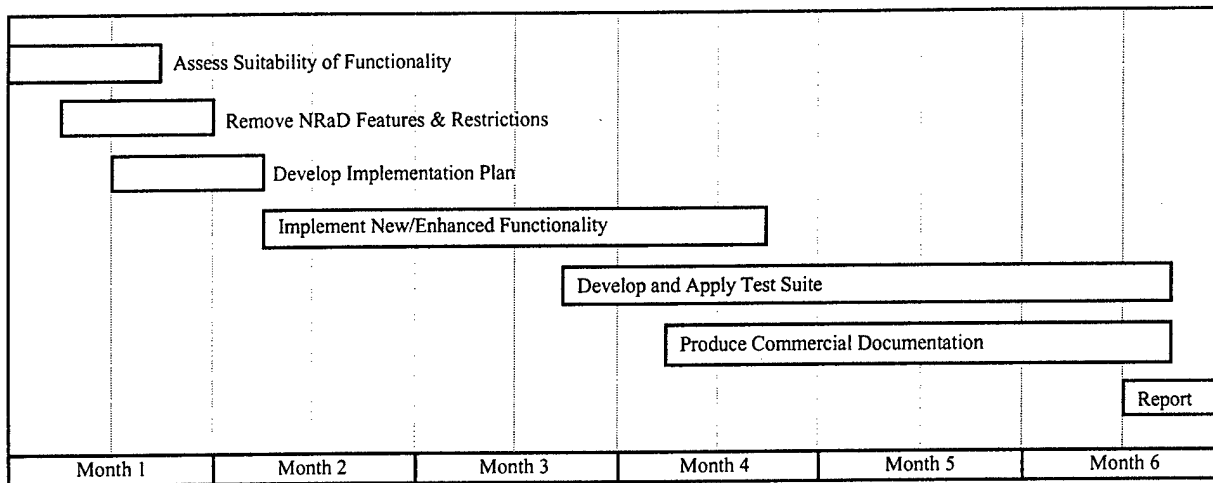
UNCLASSIFIED

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

This page intentionally blank.

## 5. Commercialization

A Commercialization Plan was provided as part of the pre-award documentation for this Phase II project. In addition, SCIENTIFIC has provided a proposal to the Navy for an optional 6-month project to carry out some of the technical work required to develop a commercial version of the TVPS. The timeline for the proposed option project is shown in Figure 3. Partial support for marketing and commercialization efforts has been obtained from Connecticut Innovations, Inc., a quasi-public Connecticut technology-funding corporation, under a Connecticut Phase III Assistance Program Agreement in the amount of \$50,000. These funds have been used to partially support a number of the initial steps towards commercialization.



**Figure 3: Phase III Time Line**

To date, we have taken the following actions to develop commercial revenues from the Phase II research and development:

1. We have retained a consultant to help us evaluate the potential of the TVPS or derivative products as a means of generating commercial revenues. This evaluation has included assessment of the current functionality as well as preliminary recommendations in regard to the most likely targets for early exploitation of a commercial version of the TVPS.
2. In the short term, it appears that there could be significant revenue generation from the TVPS by working with commercial contractors to the Department of Defense, and we are pursuing a variety of arrangements of this sort (though none are complete as of this time). Dr. Andrew Sherman, the Principal Investigator for the Phase II project, and Dr. Nicholas Carriero recently attended an ARPA meeting in Atlanta to pursue these opportunities and explore funding for related work on MPI.
3. With the assistance of our consultant, we have determined a number of additional features that would be appropriate for a commercial version of the TVPS. A partial list includes:
  - *Generalization of the data storage model in the inter-glyph VSOMs.* The present model creates a virtual file, effectively serializing the processing by downstream glyphs. In some instances (e.g., where a downstream glyph is stateless and doesn't care about the order in which discrete inputs are processed), it would, in fact, be both possible and desirable to permit multiple copies of a glyph to be executed simultaneously on

different processors. (As noted above, this would require relaxing a current restriction in Khoros.) By generalizing the data storage model to be more like a queue of discrete inputs, rather than a continuous stream of bytes, such multiple executions would become possible. This could also enable the run-time system to automatically instantiate extra copies of certain modules to overcome apparent bottlenecks in the flow of data through the Khoros glyph network.

- *Enhanced fault tolerance.* The prototype has some amount of fault resilience due to the use of the Paradise transactions. This could be enhanced and fortified in a variety of ways, including through the use of mechanisms designed to aid in reversing side effects from partial execution of glyphs.
- *Parallel glyphs.* At present, a glyph executable can be a parallel program in its own right, but it is up to the glyph developer to make certain that the inputs are properly obtained from upstream glyphs in a suitable manner. For example, if the upstream glyph is also parallel, the prototype provides no easy-to-use Cantata-level facility to transfer data between corresponding subprocesses in the two glyphs (though the developer could use VSOM technology explicitly). By exploiting the VSOM architecture and the tuple transport, however, it should be possible to specify parallel data transfer between corresponding glyph subprocesses, reserving the existing inter-glyph Khoros data flow lines for global (among the parallel components of a glyph executable) and control information.
- *Persistent glyphs.* At present, each glyph executable is reloaded each time it is used (to be consistent with standard Khoros practice). It should be fairly straightforward to create "persistent glyphs" that continue to run throughout the entire execution of the Khoros program, simply looping through successive sets of input data.
- *Enhanced data locality.* In the prototype, a single centralized Paradise server is used to store the VSOMs. This is a bottleneck, because it serializes data access, concentrates the communication load in one segment of the interconnection fabric, and tends to lead to extra communication operations. Paradise, however, supports the use of multiple servers, and it ought to be possible to exploit this capability to reduce the bottlenecks and increase communication performance and scalability.
- *Improved environment awareness.* In the prototype, the TVPS operates independently of any existing job control or system information systems that may be available. Since it is trying to exploit multiple networked resources, it would be desirable to make the TVPS work cooperatively with batch queuing systems (such as LSF [14], NQS [12], or LoadLeveler [8], for example) and to have the TVPS exploit the sort of network environment and resource information available from tools developed in projects like the SmartNet project at NRaD.

Work on some of these enhancements would be carried out under the optional project already proposed to the Navy. In addition, we have recently submitted a proposal abstract entitled "*Combining Virtual Shared Memory and Visual Programming for Metacomputing*" to ARPA for related work under their BAA 96-07.

4. We have held discussions with Hewlett Packard and Convex Computer Corporation regarding commercial versions of the technologies used in this project. Both HP and Convex are more interested in the Paradise, Piranha, and Linda products, than they are in the TVPS per se. We have a well established commercial relationship with HP, and we have a long history of working together to market SCIENTIFIC's products using such

tools as direct customer presentations, HP-sponsored industry-focused seminars, and joint booths at trade exhibitions. Our relationship with Convex is much newer, and we have focused primarily on technical product development issues, including ports of Linda and Paradise to the Exemplar, for example. Following the merger of HP and Convex, we anticipate a stronger relationship with both companies. A first step in this direction was the invitation we received from Convex to participate in the joint HP/Convex booth at SuperComputing '95 this past December in San Diego.

5. We have held discussions with Khoral Research, Inc. concerning joint commercialization of the TVPS product itself. Under the terms of our Khoros license from KRI (required to develop the TVPS and deliver it to the Navy), KRI owns all of the modifications of Khoros and is able, in principle, to commercialize the derivative work represented by the TVPS. However, KRI realizes that the Paradise and Piranha technologies would be key components of any derived commercial product, and, moreover, that SCIENTIFIC already has established marketing and sales positions in important commercial markets such as finance, petroleum, computational chemistry, among others. As a result, both companies believe that it makes most sense to work jointly to achieve commercial success.

This page intentionally blank.

## 6. Conclusions

The research and development activities have clearly met the goals of the project by developing a prototype TVPS capable of executing in parallel in a heterogeneous parallel/distributed computing environment. As we have already pointed out, the TVPS offers some significant advantages to users over either Khoros or Paradise alone, including:

1. The visual front end is easy to use, making the development of parallel programs almost transparent.
2. The fault-resilient dynamic scheduling in the TVPS means that users can be assured of successful job completion without advance designation of computation locations.
3. The use of Piranha as a glyph program executor reduces the difficulty of porting Khoros applications to new machines and operating systems.
4. The tuple transport eliminates many of the difficulties involved in heterogeneous, file-based data transfer.

We believe that a commercial version of the TVPS, which we hope to bring to market in collaboration with Khoros Research, Inc., should be of great interest to the growing community of program developers who wish to exploit parallel and distributed computing without becoming involved in the low-level implementation details.

Under the terms of the Phase II contract, SCIENTIFIC will enter into a license with the Navy to install a version of the TVPS on the Convex Exemplar and networked HP workstations at NRaD in San Diego. In addition, NRaD has requested installations for SGI workstations, a PC running Windows NT, and a Sun workstation running the Solaris operating system. SCIENTIFIC will work with NRaD to satisfy this request by installing the required Paradise and Piranha support on these systems and assisting in installation of required Khoros components (if any). (At the present time, SCIENTIFIC's modified version of Khoros runs only on the Exemplar, HP workstations, and Sun workstations running the SunOS operating system.)

UNCLASSIFIED

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

This page intentionally blank.

## 7. References

- [1] L. Cagan and A. Sherman, "Network Parallel Computing Goes Mainstream." *IEEE Spectrum* 30:12 (December), 31-35 (1993).
- [2] N. Carriero and D. Gelernter, *How to Write Parallel Programs: A First Course*. Cambridge: MIT Press, 1990.
- [3] N. Carriero and D. Gelernter, "Linda and Message Passing: What Have We Learned." Research Report, Department of Computer Science, Yale University, August 1993.
- [4] N.J. Carriero, D. Gelernter, T. Mattson, and A.H. Sherman, "The Linda Alternative to Message Passing Systems." *Parallel Computing* 20:4, 633-655 (1994).
- [5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA: The MIT Press, 1994.
- [6] D. Gelernter and D. Kaminsky, "Supercomputing Out of Recycled Garbage: Preliminary Experience with Piranha." *Proc. Sixth ACM International Conference on Supercomputing*, Washington, DC, 1992.
- [7] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA: The MIT Press, 1994.
- [8] IBM, *LoadLeveler User's Guide*, 3rd Edition, August 1995. Poughkeepsie, NY, 1995.
- [9] R. Jordan, et al, "Khoros: A Software Development Environment for Data Processing." *DSP Applications* 2:3 (1993).
- [10] D. Kaminsky, "The Piranha System for Network Computing." Research Report, Dept. of Computer Science, Yale University, 1991.
- [11] Khoros Research Inc., *Khoros 2 Manual Set* (8 Volumes). Albuquerque, NM, 1995.
- [12] B.A. Kingsbury, "The Network Queueing System — (NQS)." Draft Report, Sterling Software Inc., April 1992.
- [13] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*. Version 1.0, May 5, 1994.
- [14] Platform Computing Corp., *LSF User's Guide*, Second Edition, August 1995. Toronto, 1995.
- [15] Scientific Computing Associates Inc., *Linda User's Guide & Reference Manual* (Version 3.0). New Haven, CT, 1995.
- [16] Scientific Computing Associates Inc., *Paradise User's Guide & Reference Manual* (Version 4.0). New Haven, CT, 1996.
- [17] V.S. Sunderam, G.A. Geist, J. Dongarra, and R. Manchek, "The PVM Concurrent Computing System: Evolution, Experiences, and Trends." *Parallel Computing* 20:4 (1994).
- [18] Young, Argiro, and Kubica, "Cantata: Visual Programming Environment for the Khoros System." *Computer Graphics* 29:2 (May 1995), 22-24 (1995).

UNCLASSIFIED

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

This page intentionally blank.

## Appendix A: Piranha Documentation

This appendix contains three documents that describe Piranha and discuss its use in building adaptive parallel or distributed applications.<sup>4</sup> The documents are:

1. The UNIX *man* page for Piranha;
2. The UNIX *man* page for Piranhad, the Piranha scheduling daemon; and
3. The section entitled "Using the Piranha Model in Paradise Programs" from the *Piranha User's Guide and Reference Manual* [16].

---

<sup>4</sup> Each of the documents contains its own internal page numbers. In addition, we have added a page number of the form "A-x", centered at the bottom and beginning with A-1, for clarity in sequencing this document.

UNCLASSIFIED

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

This page intentionally blank.

## NAME

piranha – Adaptive parallel programming environment

## SYNOPSIS

*piranha-program* [*arg ...*]

## DESCRIPTION

**Piranha** is a parallel programming environment that allows idle cycles to be used on networks of workstations. When a **piranha** program is executed, that process executes the user supplied **feeder()** routine. When **piranha** processes are started on idle workstations by the **piranhad** servers on behalf of a **piranha** program, those processes execute the user supplied **piranha()** routine. Normally, the local process acts as the master, supplying tasks for the remote worker processes, and collecting the results. If a workstation becomes busy while executing a **piranha** process, the **piranhad** signals the **piranha** process to exit. This causes the user supplied **retreat()** routine to be executed by the **piranha**. The **retreat()** routine will only be executed while retreats are enabled. The **piranha()** routine calls the routines **enable\_retreat()** and **disable\_retreat()** to indicate when it is safe to retreat. It is an error to leave retreats disabled for many seconds at a time. For more information, see **piranhad(1)**.

## OPTIONS

**+FEEDER**

This is used to delimit user arguments from **piranha** specific arguments. It also indicates that this is the start of a new **piranha** job that will execute the feeder function, as opposed to a worker process that will execute the **piranha** function.

**+PIRANHA**

This is used to delimit user arguments from **piranha** specific arguments. This indicates that this is a worker process that will execute the **piranha()** function. This is only used for debugging **piranha** programs.

**-debug** <*string*>

This indicates that **piranhad** should not start **piranha** processes on behalf of this program. Instead, the user will manually start **piranha** processes, presumably using a debugger, passing this same string to each. This option corresponds to the **PIR\_DEBUG** environment variable.

**-suffixstring** <*string*>

Suffix string to append to executable file names, when needed. The default suffix string depends on the platform, and is known by **piranha** programs and **piranhad**. This is very useful for running heterogeneous **piranha** programs. For example, let's say a **piranha** program named "foo.sparc" is executed on a SunOS 4.1 system. During initialization, the **piranha** program will notice that the ".sparc" was used. It will tell the **piranhad** that its program name is "foo" with a suffix string appended. A **piranhad** on an HP will then try to execute a program named "foo.hp", and an RS/6000 will try to execute a program named "foo.rs6k". If the **piranha** program on the local machine was called simply "foo", the **piranha** program will notice that the suffix string was not used, and will tell the **piranhad** that its program name is "foo" without a suffix string. The **piranhad** will then try to execute a program named "foo", regardless of its architecture. For SunOS 4.1, the default value is ".sparc". For SunOS 5.x, the default value is ".solaris". For hpux, the default value is ".hp". For RS/6000, the default value is ".rs6k". For PowerPC, the default value is ".ppc". For SGI, the default value is ".sgi". This option corresponds to the **PIR\_SUFFIXSTRING** environment variable.

**-v/+v** Synonymous with **-verbose/+verbose**.

**-vv/+vv** Synonymous with **-veryverbose/+veryverbose**.

**-verbose**

Turns on verbose mode, which produces informational status messages.

**+verbose**

This option turns off verbose mode. This is the default.

**-veryverbose**

Turns on very verbose mode, which produces the maximum amount of informational status messages.

**+veryverbose**

This option turns off very verbose mode. This is the default.

**HETEROGENEOUS EXECUTION**

**Piranha** can be used to execute programs on a heterogeneous network of workstations. Since **Paradise** does all the necessary data conversion for tuple operations, the only difficulty is specifying the correct executable file for each workstation. There are basically two ways of doing this: executable files can be named with appropriate suffix strings, or map files can be used to equivalence different directories for different platforms. See **piranhad(1)** for more information.

**DEBUGGING**

To debug **piranha** programs, standard sequential debuggers (such as **dbx** or **gdb**) are used to execute the feeder process and each of the **piranha** processes. This is done manually, without the use of **piranhad** at all. To do this, the **+FEEDER** and **+PIRANHA** options are used so that the program knows whether it is the feeder or a **piranha**. The **-debug** option is used to specify a string that will be used to register or look up the **piranha** tuple space handle.

For example, the feeder could be started in one window with the following commands:

```
example% gdb foo
(gdb) break feeder
(gdb) run +FEEDER -debug foo_example
```

The feeder process will create a tuple space and register it with the handle server using the name "foo\_example". This name is arbitrary, but should be unique to avoid conflict with other tuple space registrations.

A **piranha** could be started in another window with the commands:

```
example% gdb foo
(gdb) break piranha
(gdb) run +PIRANHA -debug foo_example
```

The same debug string must be specified so the **piranha** will properly rendezvous with the feeder process.

Since the **piranhad** are not used, the **piranha** processes will never be signaled to retreat. To test retreating, the routine **piranha\_retreat** can be called either from the **piranha** program itself, or via the debugger (using the "call" command in **dbx** or **gdb**). This should only be called when retreats are enabled, otherwise an error will be reported. A break point should probably be put on the retreat function to avoid losing control of the process. The **piranha** process will exit as a result of executing **piranha\_retreat**.

**EXAMPLES**

Consider the following minimal **piranha** program:

```
#include <paradise.h>
extern TSHANDLE pir_ts;

int feeder(argc, argv)
int argc;
char *argv[];
{
    int i, tid, input, result;

    /* Generate the tasks */
    for (i = 0; i < NUM_TASKS; i++) {
```

```

    tid = input = i;
    out @ pir_ts("task", tid, input);
}

/* Collect the results in any order */
for (i = 0; i < NUM_TASKS; i++) {
    in @ pir_ts("result", ? tid, ? result, ? status);
    if (status == 0)
        printf("%d squared is %d0, input, result);
    else
        printf("error computing square of %d0, input);
}

return(0);
}

static int tid, input;

int piranha(argc, argv)
int argc;
char *argv[];
{
    int result;

    while (1) {
        /* Get and process the next task */
        in @ pir_ts("task", ? tid, ? input);
        enable_retreat();
        result = input * input;
        disable_retreat();
        out @ pir_ts("result", tid, result, 0);
    }
}

int retreat()
{
    /* Return task to the bag */
    out @ pir_ts("task", tid, input);

    return(0);
}

```

Since this is a Paradise program, it must include `paradise.h`. Also, there is declaration of a tuple space handle, `pir_ts`. This is automatically created by the feeder process, and automatically opened by the `piranha` processes. Although not necessary, most `piranha` programs will use this tuple space exclusively.

The `feeder()` routine creates a number of task tuples, and then collects the result tuples, printing the results. The `piranha()` routine has an infinite loop which gets a task, enables retreats, computes the task, disables retreats, and returns the results of the computation. The `retreat()` routine simply returns the task tuple, using two external static variables to communicate between the `piranha()` and `retreat()` routines.

To compile and link this program, the `cpc(1)` program is used. For example,

```
example% cpc -paradise piranha -o foo foo.cl
```

To start **piranha** program "foo" passing it the argument "10", it is executed as any standard program, as follows:

**example% foo 10**

To start **piranha** program "foo" with argument "test" and verbose messages, execute the command:

**example% foo test +FEEDER -v**

**SEE ALSO**

**piranhad(1), paradise(1), cpc(1), fpc(1)**

## NAME

piranhad – Piranha server

## SYNOPSIS

**piranhad** [*-flag ...*]

## DESCRIPTION

**Piranhad** is a server that supports adaptive parallelism using the Piranha model. It starts piranha processes on behalf of piranha programs, acting as a manager for the node that it runs on. **Piranhad** starts piranha processes when the node is idle, and signals that the piranha should retreat if the node subsequently becomes busy.

**Piranhad** is a **Paradise** application, so a **Paradise** server must be running in order to start a **piranhad**. Only one **piranhad** can be executed on a given node and **Paradise** server. Any number of **piranhad** can be started to cooperate in executing piranha jobs. Piranha programs should work properly if there is only one **Piranhad**, but there must be multiple **piranhad** to allow piranha programs to execute in parallel.

A given **piranhad** will only execute jobs that have the same uid and gid as itself unless **piranhad** is executing as root. If it is root, it will use **setuid(2)** and **setgid(2)** to become the same uid and gid as the process that started the piranha job. An authentication scheme is used to prevent **piranhad** from becoming a different user than the piranha job. For security sensitive sites, it is recommended that **piranhad** not be run as root. One possible setup is to use a special piranha account to execute **piranhad**, perhaps in a restricted environment.

While the node is idle, **piranhad** can schedule different piranha jobs to run, giving each job a time quantum to run. Currently, only one piranha job can run at a time on a given node. Any number of piranha jobs can be running across the group of **piranhad**.

If a **piranhad** attempts to execute a piranha program and fails (due to a bad uid/gid, or not finding the executable file, for example), **piranhad** will remember that the job is bad, and give up trying execute it for some period of time (about five minutes) to avoid repeatedly trying to execute the same bad job. If all **piranhad** decide that the job is bad, the piranha program will never complete.

If a **piranhad** tells a piranha process to retreat and it doesn't, the **piranhad** will respond according to the resources "retreattime", "termtimeout", and "killtimeout". These resources tell the **piranhad** how long to wait for the piranha process before taking further action. The **piranhad** can be made to signal retreat for a specified period of time, and then start sending it a SIGTERM signal. It can then start sending it a SIGKILL signal if the piranha process continues to persist. Each of these phases can be specified to be infinite, or skipped altogether. The default is to keep sending a retreat signal forever, so the terminate and kill phases will never come into play.

In addition to the command line options, a configuration file, called **piranha.config**, is used to customize the behavior of **piranhad**. It contains resource specifications which follow the rules commonly used when specifying X11 application resources. For each resource specification there is a corresponding command line option, and vice versa. The basic rules for resource specifications are explained in the "CONFIGURATION FILE" section below.

In addition to the configuration file, **piranhad** also uses map files to handle differences between the file systems on the different nodes that the **piranhad** are running on. See the "MAP TRANSLATION" section for more information.

**Piranhad** should not be used when debugging piranha programs. To debug a piranha program, it should be started manually using the **-debug** option. See **piranha(1)** for more information.

## OPTIONS

The following options each correspond to a resource/value pair of the same name in the configuration file, except as noted. See the "CONFIGURATION FILE" section for a description of the format used by the configuration file.

**-advance** *<float>*

Load average below which the piranha process will advance. This value should be more than one less than the value specified by the **-retreat** *<float>* option, otherwise the **piranhad** may cycle continuously between being available and unavailable. The default value is 1.9.

**-advancecheck** *<seconds>*

Number of seconds to wait between checks to advance. Setting this value to a very small value could cause **piranhad** to consume a significant amount of CPU time, and thus annoy the user of the workstation. The default value is to check every 10 seconds.

**-enabled** never | check | always

Determines when piranhas can use the node. The literal value "never" means that piranha can never use the node. The value "check" means that piranha can use the node when it is idle. The value "always" means that piranha can always use the node, regardless of idleness. The default value is "check".

**-host** *<server-host>*

Name of the host running a Paradise server. The default is to use the host specified by the **PARADISE\_HOST** environment variable, or the local host, if **PARADISE\_HOST** is not defined.

**-idle** *<seconds>*

Number of seconds that the node must be idle before it can be used by a piranha. To determine idleness, **piranhad** checks when various devices were last used. These devices include such things as the console keyboard, the mouse, and pseudo-terminals, although not all of these are supported on all platforms. A value of zero means to ignore user idleness, however load average may still be checked. The default value is 300 seconds (5 minutes).

**-jobcheck** *<seconds>*

Number of seconds to wait between checks for a waiting job while the workstation is idle and available. Since the node is idle, this value can be relatively small without annoying anyone. The default value is 10 seconds.

**-killtimeout** *<seconds>*

Number of seconds to wait for a piranha to die after starting to send the piranha a SIGKILL. A value of zero means to never timeout, and a value of -1 (or any negative value) means to skip the kill phase. The default is to skip the kill phase (as indicated by a value of -1).

**-loadperiod** 1 | 5 | 15

Load average period: 1, 5, or 15. The default value is 5, for 5 minute load averaging.

**-quantum** *<seconds>*

Minimum number of seconds allotted for each piranha job. After a piranha process has executed for this period of time, the **piranhad** may ask the current piranha process to retreat, and it will then schedule a new piranha process to run. The default time quantum is 300 seconds (5 minutes).

**-retreat** *<float>*

Load average above which piranhas retreat. This value should be more than one greater than the value specified by the **-advance** *<float>* option, otherwise the **piranhad** may cycle continuously between being available and unavailable. The default value is 3.0.

**-retreatcheck** *<seconds>*

Number of seconds to wait between checks to retreat. Setting this to a large value could prevent the piranha from retreating in a timely fashion. The default value is 10 seconds.

**-retreattimeout** *<seconds>*

Number of seconds to wait for piranha to retreat. A value of zero means to never timeout, but continuously signal a retreat until the piranha finally dies. The default is never timeout (as indicated by a value of 0).

**-suffixstring** *<string>*

Suffix string to append to executable file names, when needed. The default suffix string depends on the platform, and is known by piranha programs and **piranhad**. For SunOS 4.1, the default value is ".sparc". For SunOS 5.x, the default value is ".solaris". For hpux, the default value is ".hp". For RS/6000, the default value is ".rs6k". For AXP/Alpha, the default value is ".alpha". This option is very useful for running heterogeneous piranha programs. See the "HETEROGENEOUS EXECUTION" section for more information.

**-termtimeout** <seconds>

Number of seconds to wait for piranha to terminate after starting to send the piranha a SIGTERM. A value of zero means to never timeout, but continuously try to terminate the piranha until it finally dies. A value of -1 (or any negative value) means to skip the terminate phase. The default is to skip the terminate phase (as indicated by a value of -1).

**-v/+v** Synonymous with **-verbose/+verbose**. Note that there is no "v" resource defined by the configuration file. The "verbose" resource must be used instead.

**-vv/+vv** Synonymous with **-veryverbose/+veryverbose**. Note that there is no "vv" resource defined by the configuration file. The "veryverbose" resource must be used instead.

**-verbose**

Turns on verbose mode, which produces informational status messages. This is equivalent to setting the "verbose" resource to true.

**+verbose**

This option turns off verbose mode. This is the default. This is equivalent to setting the "verbose" resource to false.

**-veryverbose**

Turns on very verbose mode, which produces the maximum amount of informational status messages. This is equivalent to setting the "veryverbose" resource to true.

**+veryverbose**

This option turns off very verbose mode. This is the default. This is equivalent to setting the "veryverbose" resource to false.

## CONFIGURATION FILE

A configuration file contains resource specifications used to customize the behavior of **piranhad**. **Piranhad** looks for two different configuration files. They are, in order of precedence:

- 1) A user configuration file, **.piranha.config**, located in the user's home directory.
- 2) The global configuration file located in the Paradise tree in **common/lib/piranha.config**.

A default global **piranha.config** file is supplied with Paradise. It may be customized by a system administrator to suit your requirements. Resources defined in the user's **.piranha.config** take precedence over the global **piranha.config** file.

Each line in the configuration file, not beginning with the comment "!" character, is called a resource definition. In order for **piranhad** to recognize a resource definition, the definition must use the following format:

```
programspec.nodespec.userspec.resourcespec: value
```

where:

**programspec**

is either the class name "Tsnet", or a specific instance of this class, ie. **piranhad**.

**nodespec**

is the class "Node", or a specific node name, such as "myhp9000".

**userspec** is either the class name "User", or a specific user name, such as "frank".

**resourcespec**

is a variable name recognized by **piranhad** which can be assigned values.

**value** is the value assigned to the resource.

If an incorrect format is used, the resource definition will be ignored by **piranhad**. Beware of spelling errors. For example:

```
! This is correct and will be used
piranhad.Node.User.retreat: 10.0
```

```
! This is incorrect and will be silently ignored
piranhad.Node.Usr.advance: 8.0
```

Programspec, nodespec, userspec, and resourcespec are called resource definition components. The first three have a single class name associated with them that can be used like a wildcard. Class names begin with an uppercase letter by convention, and instance names begin with a lower case letter. Currently, there is no class name associated with the resourcespec component.

Class names work as wildcards. By using the class name, you can quickly set resource values for all members of a class. Using instance names you can set resource values for a specific instance of that class. Instance names take precedence over class names.

## MAP TRANSLATION

Map translation is a way of defining equivalences between local directory trees and directory trees on remote nodes. It is used by **piranhad** to determine both the execution directory and working directory. If you are running piranha programs from within directories that have identical path names on all the nodes running **piranhad**, then map translation is not needed. But if the directories are mounted with different path names, then map translation can be very useful for allowing the **piranhad** to find piranha executables and the proper working directories.

Map translation has nothing to do with physically copying program executables between different machines. It is only used to translate names so they can be correctly located. Quite often, NFS is used to share the executables between multiple machines, but that doesn't guarantee identical path names, particularly if automounting is used.

Map translation is controlled by the rules put into both the global *common/lib/tsnet.map* (relative to the Paradise installation tree) and the local *~/tsnet.map* file. If no translation is available for a given directory and/or node (whether because its rules haven't been specified or no map translation file exists), the rule is simple: look for the same directory on the remote node. For example, if the specific local executable directory is */usr/bin/piranha*, then the generic directory is */usr/bin/piranha*, and the remote executable directory used for each **piranhad** is */usr/bin/piranha*.

Map translation file entries use the following commands:

**mapto** Map a specific local directory to a generic directory.

**mapfrom** Map a generic directory to a specific directory (usually on a remote node).

**map** Equivalent to a **mapto** and a **mapfrom**, mapping specific directories to and from a generic directory.

If a local and/or global *tsnet.map* file exists, **piranhad** uses **mapto** rules to transform a local directory path into a generic directory path, and **mapfrom** rules to transform the generic directory path into specific remote directories. As we'll see later, a **mapto** does not need a corresponding **mapfrom**, nor does a **mapfrom** need a corresponding **mapto**. A **map** is a **mapto** and **mapfrom** rolled into one. This will be explained in detail below.

```
mapto generic {
  local-node: local-specific-dir;
  etc...
}
```

```
mapfrom generic {
  remote-node1: remote-specific-dir1;
```

```

remote-node2: remote-specific-dir2;
etc...
}

```

```

map generic {
  (remote or local)-node: (remote or local)-specific-dir;
  (remote or local)-node2: (remote or local)-specific-dir2;
  etc...
}

```

The **mapto** works as follows. The **piranha** program takes the local node name and the local directory, and matches it against each "local-node: local-specific-dir" pair in the {} section of a mapto. It looks for the longest match. For example if the local directory on node moliere was /u/person/programs, and the tsnet.map contained:

```

mapto /net/moliere {
  moliere : /;
}

```

```

mapto /net/moliere/tmp {
  moliere : /u/person
}

```

The match would be "moliere : /u/person", and the generic directory would become /net/moliere/tmp/programs. Note that the unmatched portion of the local directory, programs, is appended to the mapto's generic directory name. If only the 1st mapto was in the tsnet.map file, the match would have been "moliere : /", and the generic directory would have been /net/moliere/u/person/programs. If no match is found in the user's .tsnet.map file, the global tsnet.map file is searched. If a match is found in the user's .tsnet.map file, the global tsnet.map is not searched even if it contains a longer match.

In the example above with 2 mapto's, and no mapfrom's, every **piranhad** will use the directory /net/moliere/tmp/programs. Since there were no mapfrom's, the generic directory was null transformed (without change) to remote specific directories.

Though a mapto may contain a list of "local-node: local-specific-dir" lines; only those where the local-node is the same as the node from which the piranha program is invoked are used to match the local directory. It is simpler to understand mapfile translation if you imagine that a mapto only contains lines that match the local node name. This highlights mapfile translations one to many property. One local specific directory is transformed to a generic directory which is finally transformed to many specific remote directories (one for each remote node on which piranhad is executing). But in reality, it is useful to write mapto's that contain lines with various different local-node names. Especially if your home directory is mounted on many different nodes, this will make your .tsnet.map file useful when running piranha programs on all of those different nodes.

If the piranha program does not find any matching mapto "local-node : local-specific-dir" pair in either the local or global tsnet.map file, no transformation occurs and the generic directory becomes the same as the local directory. This is called a null transformation.

Next, the **piranhad** must transform the generic directory back into remote-specific directories for their own node. They do this using **mapfrom**. If the **piranhad** do not find any matching mapfrom generic directory, the generic directory is used as the remote specific directory on each remote node. This is again a null transformation, and is what occurred in the example above. A mapto with no matching mapfrom generic directory is very common, especially in networks where shared directories are consistently mounted in identical locations on every node.

A mapfrom contains a list of "remote-node: remote-specific-dir" pairs. There is often a pair for each remote node on which there is a **piranhad**.

With a mapfrom, **piranhad** matches the generic directory name which is right after the 'mapfrom' keyword to the generic directory name it created via a mapto (or to the local directory name itself if there was no mapto and therefore a null transformation). It then searches the {} portion of the mapfrom looking for each remote-node that matches the names of the remote node on which it is trying to start a piranha process. If it finds a match, it uses that specific directory in that "remote-node : remote-specific-dir" pair as possibly the remote node's working and/or executable directory. Only if no match is found in the user's local .tsnet.map file, does **piranhad** search for a match in the global tsnet.map file.

If no matching mapfrom is found, the generic directory path is null transformed and is used as is.

A map is a mapto and mapfrom rolled into one. A map therefore must always contain a "local-node :local-specific-dir" entry for the node on which **piranhad** is run. Just like a mapto had to contain an entry for the local-node. A map also contains "remote-node :remote-specific-dir" pairs just like those in a mapfrom. Since the generic directory is always translated to a specific remote directory before being used by **piranhad**, the generic directory name used in map, mapto and mapfrom need not even be a real directory. Of course if you use a mapto without a matching mapfrom, or a mapfrom without a matching mapto, the generic directory name needs to be a real directory. This is due to the null transformations discussed above. If a null transformation occurred on a generic non-real directory name, **piranhad** would try to use that non-real directory when starting the program, and of course it would fail because the non-real directory doesn't exist.

example:

```
map special {
  willow : /root/homes/myhome/test_suite/willow;
  oak    : /root/homes/myhome/test_suite/oak;
  maple  : /root/homes/myhome/test_suite/maple;
}
```

If the local node is willow, and the local directory is /root/homes/myhome/test\_suite/willow, the remote specific directory for a **piranhad** on oak will be /root/homes/myhome/test\_suite/oak, and the remote specific directory for a **piranhad** on maple will be /root/homes/myhome/test\_suite/maple. This usage is useful if you want each remote node to use a different working directory so that output files for each piranha process won't conflict.

Another useful map example is:

```
map /tmp {
  willow : /root/homes/myhome/test_suite/willow;
  oak    : /root/homes/myhome/test_suite/oak;
  maple  : /root/homes/myhome/test_suite/maple;
}
```

Again if the local node is willow, and the local directory is /root/homes/myhome/test\_suite/willow, the remote specific directory for **piranhad** on oak and maple is specifically described as above by the contents of the map. If a piranha process is started on any other remote node, its remote specific directory will be /tmp. Since specific rules didn't exist in the map for nodes other than oak and maple, a null transformation is done, and the generic name is used as the remote specific name. This is a case where you would want the generic directory name in the map to be a meaningful directory name.

The best way to understand the power of mapfile translation is by example:

example1:

This is a common mapto example when remote nodes automount your node's directory. On your node, moliere, you refer to your piranha\_tests directory as /u/people/mine/piranha\_tests. On remote nodes, you refer to your piranha\_tests directory on moliere as /net/moliere/u/people/mine/piranha\_tests. In this

case, the piranha program needs the following mapto in .tsnet.map.

```
mapto /net/molieres {
    molieres : /;
}
```

To explain our first example in more detail, let's say that you're executing a piranha program called /u/people/mine/piranha\_tests/rayshade.net. You're on your node molieres and in your /u/people/mine/piranha\_tests directory. When the piranha program starts executing, it must tell the **piranhad** where they can find rayshade.net. If it told the remote nodes that rayshade.net was in /u/people/mine/piranha\_tests, they would not be able to find it, because remote nodes mount molieres:/u/people/mine/piranha\_tests as /net/molieres/u/people/mine/piranha\_tests. The mapto solves this problem. The piranha program looks in the {} section of the mapto and sees that there is a line for molieres. It then sees if the local-specific-dir in that line of the mapto "/" matches the directory where the local rayshade.net executable resides. It does, so the piranha program uses the generic directory "/net/molieres" prepended to the unmatched portion of the local directory "u/people/mine/piranha\_tests", to form a generic directory /net/molieres/u/people/mine/piranha\_tests. Since there is no mapfrom with a generic directory that matches /net/molieres/u/people/mine/piranha\_tests, the mapfrom is a null transformation, and /net/molieres/u/people/mine/piranha\_tests is used by all of the **piranhad**.

At this point, it may be wise to note that all **piranhad** attempt the mapfrom translation, even if it is running on the same node as the piranha program. In the example above, even a process started on the local node will be started by **piranhad** using the translated directory /net/molieres/u/people/mine/piranha\_tests. So in this case, even molieres must be able to refer to /u/people/mine/piranha\_tests as /net/molieres/u/people/mine/piranha\_tests.

If you wanted to have workers started on your local node molieres use /u/people/mine/piranha\_tests rather than the translated directory, you could use a mapto/mapfrom as follows:

```
mapto /net/molieres {
    molieres : /;
}
mapfrom /net/molieres {
    molieres : /;
}
```

#### example2:

```
mapto /net {
    molieres: /tmp_mnt/net;
}
```

This mapto is useful when your local directory is actually on another node that's automounted by your node. In this example, when on molieres in /net/chaucer/overflow/tests, pwd actually displays your local directory name as /tmp\_mnt/net/chaucer/overflow/tests. The problem that this mapto solves, is that /net/chaucer/overflow/tests is never referred to as /tmp\_mnt/net/chaucer/overflow/tests even though /tmp\_mnt is the mount point that the automounter uses. So the /tmp\_mnt portion of the directory must be stripped off. That is just what this mapto accomplishes.

#### example3:

```
map /usr/bin/piranha {
    molieres : /usr/local/bin/piranha;
    chaucer : /usr/tmp;
    sappho : /usr/bin/piranha;
}
```

This map specifies equivalence directories explicitly for the three named nodes in the {} section of the map, and implicitly for every other node in your network. For example, if the piranha program is invoked on molieres and the executable is located in /usr/local/bin/piranha, this map will cause the

**piranhad** on chaucer to look for that executable in /usr/tmp; the **piranhad** on sappho to look for the executable in /usr/bin/piranha (note that the translation for sappho is redundant since this is the default), and in /usr/bin/piranha on every other node in your network (a null transformation is performed in the mapfrom portion of the map for any node that is not explicitly listed in the {} section of the map).

Mapfile Translation Wildcards:

There are two wildcard characters allowed in map translation files entries:

- 1) An asterisk (\*) can be used as a wildcard in local and remote node names.
- 2) An ampersand (&) can be used to substitute the current node name within a translation.

#### HETEROGENEOUS EXECUTION

Let's say a piranha program named "foo.sparc" is executed on a SunOS 4.1 system. During initialization, the piranha program will notice that the ".sparc" was used. It will tell the **piranhad** that its program name is "foo" with a suffix string appended. A **piranhad** on an HP will then try to execute a program named "foo.hp", and an RS/6000 will try to execute a program named "foo.rs6k". If the piranha program on the local machine was called simply "foo", the piranha program will notice that the suffix string was not used, and will tell the **piranhad** that its program name is "foo" without a suffix string. The **piranhad** will then try to execute a program named "foo", regardless of its architecture.

#### EXAMPLES

Normally, **piranhad** will be executed in the background. The main exception is when verbose messages are desired. Therefore, all the following examples use an ampersand to run **piranhad** in the background except the verbose example.

To start **piranhad** with verbose messages, execute the command:

```
example% piranhad -v
```

To start **piranhad** so that it uses the Paradise server on node "mysparc":

```
example% piranhad -host mysparc &
```

The -host option is particularly useful when executing **piranhad** remotely via **rsh**, for example.

If **piranhad** should have exclusive use of a node, the following command will turn off retreats:

```
example% piranhad -enable always &
```

To run **piranhad** so that it retreats at or above a load average of 4, starting again at a load average of 2.5, use the following command:

```
example% piranhad -retreat 4 -advance 2.5 &
```

To make **piranhad** respond very quickly to a node becoming busy, use the command:

```
example% piranhad -retreatcheck 2 &
```

#### SEE ALSO

**piranha(1)**, **paradise(1)**, **cpc(1)**, **fpc(1)**

#### BUGS

**Piranhad** only supports one piranha process per node, even if the node is a multiprocessor.

There is no mechanism to cause the default piranha tuple space to be created on a different Paradise server than the one containing the **piranhad** tuple spaces.

## Using the Piranha Model in Paradise Programs

This section presents the Piranha model of distributed computing. Piranha was motivated by the observation that most networked workstations are idle the majority of the time. These wasted cycles represent a large potential source of computation, and if they are used carefully, one can do so without impacting the interactive users of the machines. Piranha was developed to do just that.

At any given time, Piranha declares machines as either idle or busy, depending on such criteria as load average or keyboard or mouse activity. When machines become idle, they can join a Piranha computation. When they become busy again, for example because the owner of the machine begins typing, the machine will immediately cease work on the Piranha computation.

Piranha programs often share many characteristics with other Paradise programs. They use tuples and tuple spaces for communication and synchronization, and much of the preceding discussion in this manual will apply to them. The programs most suited to being expressed in Piranha are master/worker algorithms with minimal data dependency between tasks. However, it is possible to write Piranha programs that have significant inter-task dependencies. See the LU factorization case study later in this section for an example of an application with strong synchronization requirements.

### Piranha Program Structure

All Piranha programs are required to provide three procedures: `feeder`, `piranha`, and `retreat`.

#### `feeder`

The `feeder` routine generally functions as the master, and is invoked on the local node when a Piranha program begins execution. It is responsible for generating tasks for and collecting results from the worker processes. The process running `feeder` is special in that it never retreats; in fact, the Piranha system will terminate program execution when `feeder` returns. `feeder` is passed the invocation (command line) arguments when it is invoked.

#### `piranha`

The `piranha` routine is executed by the worker processes created by the Piranha facility. This function is often written as an infinite loop, each iteration of which processes a single task. This routine will execute until it completes or it is forced to vacate the node on which it is running. In the latter case, the Piranha system automatically calls the `retreat` function and exits.

### **retreat**

The `retreat` routine is what allows a `piranha` process to discontinue executing without affecting overall program integrity or results. At a minimum, it is responsible for returning any unfinished (or partially finished) tasks to tuple space, to be retrieved by some other `piranha` process. More complex versions also provide needed data and state information for partially finished tasks, which will allow a different `piranha` process to take them up at the point where they left off, rather than having to restart them from the beginning.

### **enable\_retreat and disable\_retreat**

The Piranha system also provides two additional functions: `enable_retreat` and `disable_retreat`. These routines allow critical sections of the program to be protected against retreats. Retreats *must* be disabled whenever Paradise operations are executed.

`enable_retreat` indicates the beginning of a program section where retreats are allowed. Until the first invocation of this operation, retreats are disabled within a Piranha program.

`disable_retreat` prevents retreats, marking the end of the program section begun with the `enable_retreat` operation. Looked at another way, `disable_retreat` marks the beginning of a protected region of the program, during which retreats are not allowed.

### **Program Termination**

A Piranha program terminates when `feeder` returns or exits. When this happens, all running `piranha` processes will be terminated automatically.

### **A Simple Piranha Program**

The following C program will demonstrate all of the Piranha constructs. It can be used as a template to create your own Piranha programs:

```
#include <paradise.h>
TASK *task, *get_task();
RESULT f();
int index;
extern TSHANDLE pir_ts; /* tuple space provided by Piranha */

feeder(argc, argv)
int argc;
char **argv;
{
    RESULT result;
    int count, process_result();
```

```

/* create task tuples in the pir_ts tuple space */
for (count=0; task=get_task(count); ++count)
    out @ pir_ts("task", count, *task);
/* retrieve results from the pir_ts tuple space */
while (count-->0) {
    in @ pir_ts("result", ?index, ?result);
    process_result(index, result);
}

piranha()
{
    RESULT result;

    while(1) {
        in @ pir_ts("task", ?index, ?*task);
        enable_retreat();
        result = f(task); /* f performs the actual computation */
        disable_retreat();
        out @ pir_ts("result", index, result);
    }
}

retreat()
{
    /* return unfinished task to the pir_ts tuple space */
    out @ pir_ts("task", index, *task);
}

```

`get_task` returns a pointer to a task structure, or `NULL` if there are no more tasks. `f` is the function that we want to invoke in parallel; it performs the real work of the program. Notice that both `index` and `task` are global variables. This is necessary in order to make them accessible to the `retreat` function.

These routines all use the “`pir_ts`” tuple space, which is provided automatically by the Piranha facility. In order to use this tuple space, all that is necessary is to include `paradise.h` and to declare it as an `extern TSHANDLE`. If the program wanted to access any other tuple space, it would have to create one or retrieve and open a tuple space handle in the normal manner.

## Building Paradise Piranha Programs

The preceding program could be compiled and linked with a command like the following:

```
$ cpc -paradise piranha -o fish fish.cl
```

The option to `cpc` indicates that this Paradise program uses the Piranha facility.

## percolate: A Monte Carlo Simulation

The following C program is a more advanced example of Piranha functionality. It is a Monte Carlo simulation named **percolate**. Here is a simplified version of its feeder function. The program begins by processing its arguments, setting some global parameters, and placing them into tuple space:

```
#include <paradise.h>
extern TSHANDLE pir_ts;

feeder(argc,argv)
int argc;
char **argv;
{
    /* set up global parameters & put into tuple space */
    set_params(argc, argv, &params);
    out @ pir_ts("params", params, X_STEP);
```

feeder's main **for** loop creates tasks and gathers results, each within their own **for** loop:

```
    /* loop over series */
    for (x=params.gmax_x; x <= MAX_X; x += X_STEP) {
        tasks_out = 0; /* reset for each series */

        /* create tasks, being careful not to flood TS */
        for (tasks_out=0; tasks_out < WATERMARK &&
            tasks_out < NUM_TRIALS; tasks_out++)
            out @ pir_ts("task", x, tasks_out+MIN_SEED);

        /* gather results; put out more tasks if approp. */
        for (i=0; i < NUM_TRIALS; i++) {
            in @ pir_ts("result", x, ?result);
            data_out(&result, &list, NUM_TRIALS);
            if (tasks_out < NUM_TRIALS) {
                out @ pir_ts("task", x, tasks_out+MIN_SEED);
                tasks_out++;
            }
        }

        out @ pir_ts("task", x, NEXT_X); /* go on to next series */
    }
} /* end of feeder */
```

The program performs the simulation **MAX\_X** times; each iteration of the outermost loop corresponds to one series of trials, with a distinct value for the variable **x**. Within each series, this feeder routine creates **NUM\_TRIALS** tasks. The second **for** loop creates up to **WATERMARK** tasks; if **WATERMARK** is less than **NUM\_TRIALS**, the remaining tasks are created in the second **for** loop as the "result" tuples are gathered. This watermarking technique is used to avoid filling

up tuple space with the large number of task tuples required by this program. The reason we **out** `tasks_out+MIN_SEED` in addition to `x` is to give each task a unique, non-trivial seed for the random number generator.

Once all of the results for the given series of trials are retrieved, the program places a "task" tuple into tuple space with the special value `NEXT_X` as its third argument. This will tell the piranha (worker) processes that a new series is beginning.

The piranha routine for this program begins by reading in the global parameters and assigning an initial value to the variable `x`:

```
void piranha()
{
    rd @ pir_ts("params", ?params, ?x_step);
    x = params.gmax_x;
```

Next, `piranha` enters an infinite **while** loop, and retrieves the next task from tuple space:

```
while (1) {
    in @ pir_ts("task", x, ?seed);

    /* new series */
    if (seed == NEXT_X) {
        /* put task back for others to use */
        out @ pir_ts("task", x, seed);
        /* increment gmax_x and reassign x */
        params.gmax_x += x_step;
        x = params.gmax_x;
    }

    else {
        enable_retreat(); /* allow retreats */
        percolate(seed, &params, &result);
        disable_retreat(); /* no retreats anymore */
        out @ pir_ts("result", x, result);
    } /* end if-then-else */
} /* end while */
} /* end piranha */
```

The first section of the **if-then-else** construct handles the special case task for a new series; the second section performs the actual computation. The real work is done by the routine `percolate`, which essentially corresponds to the original sequential program (minus argument handling and global parameter setting). While `percolate` is running, retreats are enabled, and the Piranha system can shut down the `piranha` process. At all other times, retreats are disabled.

Here is the retreat function for this program:

```
retreat()
{
    out @ pir_ts("task", x, seed);
}
```

This simple function simply places the current task back into tuple space, where some other piranha process will retrieve it and perform it in its entirety. Note that the variables `x` and `seed` must be globals.

## Driving a External Program with Piranha

The next example illustrates the use of a Paradise Piranha program to initiate and control multiple instances of an existing simulation program (whose command invocation string is stored in the array `task_string`). Here are the global declarations and the feeder routine for this program:

```
#include <stdio.h>
#include <paradise.h>
extern TSHANDLE pir_ts;

TSHANDLE pts;
int task_active, task_num;
char task_string[25];

/* feeder simply registers the piranha TS handle,
 * and then waits for a shutdown request. */
feeder()
{
    register_handle("Piranha TS", "Simula2", "Piranha", &pir_ts);
    in @ pir_ts("SHUTDOWN");
    deregister_handle("Compute TS", "Simula2", "Piranha");
    return;
}
```

This routine registers the handle for the Piranha tuple space, and then it waits for a "SHUTDOWN" tuple to appear in that tuple space, whereupon it deregisters the handle and exits.

Here is the piranha routine:

```

/* piranha executes one task, structured as a transaction;
 * if it retreats, the transaction is cancelled. */
void piranha()
{
    FILE *fp;

    while (1) {
        task_active = 0;

        xaction @ pir_ts(PARADISE_TIMEOUT(600));
        task_active = inp @ pir_ts("sim2 task", ?task_num,
                                   ?task_string:);
        if (task_active) { /* We found a task, so do it. */
/* Create the task tuple that will be used by the actual
 * application code; a separate handle to the "pir_ts" TS
 * must be created and used to distinguish it from the one
 * used in the xaction operation. */
            pts = restrict @ pir_ts(PERM_ALL);
            open @ pts();
            out @ pts("task", task_num);

            enable_retreat();
/* Run the simulation program via the system function. */
            if (status = system(task_string)) {
                perror("command failed"); /* command failed */
                additional print statements
            }
            disable_retreat();

            task_active = 0;
            out @ pir_ts("task done", task_number);
            close @ pts();
            commit @ pir_ts();

        } /* end if(task_active) */

        else { /* There was no task, so just end the transaction. */
            commit @ pir_ts();
            enable_retreat();
            sleep(10); /* Wait before getting next task. */
            disable_retreat();
        } /* end else */
    } /* end while */

    return;
} /* end piranha */

```

The heart of the routine's while loop is structured as a transaction, including the **inp** operation which attempts to retrieve a task to complete; the task tuple will not be permanently removed from the tuple space until the transaction is committed, so it will still be available for another piranha process if one that begins it is forced to retreat.

When a task is successfully retrieved, the function copies the tuple space handle for the Piranha tuple space and opens it (this is done so as to not conflict with the handle used in the transaction). It creates another tuple, the "task" tuple, for use by the actual simulation program, and then runs it via the **system** system call. It creates a "task done" tuple and commits the transaction after the program completes (regardless of its outcome). Note that retreats are allowed only while the external program is running.

When it cannot find a task tuple, it commits the current transaction, waits 10 seconds, and then tries again via the next iteration of the while loop.

Here is the retreat function:

```
void retreat()
{
    if (task_active) {
        inp @ pts("task", task_number); /* Remove generated tuple. */
        close @ pts();
    }

    cancel @ pir_ts();
    return;
}
```

If **task\_active** is true, indicating that a task was retrieved, then the routine removes the "task" tuple from the Piranha tuple space (if present) and closes the (second) tuple space handle. It then cancels the pending transaction.

The actual computation code is a Fortran program. Here is the main routine, which is what gets initiated by the **piranha** routine's system call:

```
      Program Real_Comp
c
c This is the program executed by piranha. It gets some basic (common
c block type) data and a task number. It then calls the real compute
c program (which is basically identical to the sequential version).
c
      TSHANDLE pts
      integer task_num, fwaitfor_handle
      common /dbldat/ a, b, c
      common /intdat/ i, j, k
c
```

```

c Obtain and open the Piranha tuple space.
  istat = fwaitfor_handle('Piranha TS', 'Simula2', pts)
  open @ pts()
c
c Read paramaters from tuple space and save into common blocks.
  rd @ pts ('dbldat',?/dbldat/)
  rd @ pts ('intdat',?/intdat/)
c
c Grab a task and perform the computation. task_num is the task
c identifier for the main computation routine, identifying its associated
c data file.
  in @ pts('task', ?task_num)
  Call main_comp(task_num)
c
  close @ pts()
  end

```

main\_comp is essentially the original sequential program.

All that remains is the driver subroutine which is called from the front-end to the original program, which translates a user's request into a Paradise task:

```

Subroutine pinvoke(strw, ntasks)
c
c Called from main computation controller (the user GUI)
c with the basic UNIX command line to be executed for each task.
c
  integer fwaitfor_handle
  character*80 strw
  TSHANDLE pts
  common /dbldat/ a, b, c
  common /intdat/ i, j, k
c
c Get tuple space handle and out global common blocks.
  istat = fwaitfor_handle('Piranha TS', 'Simula2', pts)
  open @ pts()
  out @ pts ('dbldat', /dbldat/)
  out @ pts ('intdat', /intdat/)
c
c Create task tuples for the Paradise program.
  do n = 1, ntasks
    out @ pts('task invocation string', n, strw:)
  enddo
c
c Retrieve tuples indicating completed runs.
  do n = 1, ntasks
    in @ pts('task done', ?m)
  enddo
c

```

```

c Shut down Paradise program and exit.
  out @ pts('SHUTDOWN')
  close @ pts()
  return
end

```

This routine can be called as necessary by the graphical interface to the simulation program. It places some global dat into the Piranha tuple space, and then generates task tuples for the C Paradise program. It then retrieves “task done” tuples as they appear, and ends by placing the “SHUTDOWN” tuple into the Piranha tuple space, which tells the Paradise program to exit as well.

## The Piranha Configuration File

The Piranha system has its own system-wide configuration file, *piranha.config*, located in */etc* or in the *common/lib* subdirectory of the Paradise tree. The */etc* location takes precedence; if a configuration file exists in */etc*, a file existing in *common/lib* will not be used. A user-specific configuration file *.piranha.config* may also be used, located in the user's home directory, and the two files are logically merged at runtime (precedence is given to the user-specific file in the case of conflicts).

These configuration files specify the conditions under which execution can proceed on the various available nodes.

The entries in the Piranha configuration file are of the form:

*piranha.nodespec.userspec.resourcespec: value*

where *nodespec* is the class *Node* or the name of an individual node, *userspec* is either the class *User* or a specific username (on the local node), and *resourcespec* is the name of a resource (available resources are discussed individually below), which is to be assigned the specified value.

The **enabled** resource determines whether Piranha may run on a node at all. The resource may be set to always, check, or never. The default value is check.

When it is allowed to run on a node, the Piranha system looks at system load average and device idle times when deciding whether or not to start a *piranha* process on a node; the same considerations determine when a process needs to retreat. The **idle** resource specifies how long user devices (such as the keyboard) must be idle before the Piranha system can use that node (in seconds, with a default value of 300).

The **retreat** resource specifies how high the load average on the system may go before a running *piranha* must retreat. Its value is a floating point number and defaults to 3.0. The **loadperiod** resource specifies the number of minutes over which the load average is computed (the default value is 5). The **retreatcheck** resource specifies how often a *piranha* should check if it should retreat while it is running (in seconds); the default value is 10.

Once a Piranha process has retreated, it may still become active again provided that system load decreases sufficiently. The condition specified by the **idle** resource must be fulfilled, as well as that imposed by the **advance** resource. The latter specifies how low the load average must drop before the piranha may advance (resume execution). It is designed to prevent the piranha's retreat from reducing the load average sufficiently to allow it to immediately restart. The default value is 1.9. The **advancecheck** resource specifies how often a suspended Piranha process should check if conditions allow it to advance or not (in seconds); the default value is 10.

See chapter 7 for a complete list of Piranha configuration file resources.

Here are some sample Piranha configuration file entries:

```
piranha*molire*chavez*idle: 600
piranha*molire*chavez*retreat: 4.0
piranha*molire*chavez*advance: 2.5
piranha*moleire*chavez*retreatcheck: 5
piranha*molire*User*retreat: 2.5
```

The first four lines apply only to user *chavez* on the node *molire*. They set the idle time required before a piranha executes to 10 minutes. A Piranha process must retreat when the load average rises above 4.0, and it cannot advance until it falls below 2.5. A running process must check conditions every 5 seconds. The final line applies to all other users on *molire*, and it indicates that a Piranha process must retreat whenever the load average rises above 2.5.